Cairo University
Faculty of Engineering
Department of Computer Engineering

Inferex

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by
Waleed Ousama Khamees

Supervised by
Dr. Sandra Wahid

July / 2025

Abstract

Inferex is an intelligent desktop application designed to help users recall and retrieve information they have previously seen on their computers, whether from a video, document, or website, without the need to manually save or organize it. In today's digital world, users interact with large volumes of information every day, yet remembering specific details from past activities remains a constant challenge. Most AI assistants currently available require users to manually provide the context of the question in order to receive an accurate and personalized answer, which limits their convenience and effectiveness. Inferex solves this problem by silently and privately building a local context from the user's screen activity, enabling it not only to retrieve previously seen content but also to answer related questions intelligently without requiring additional input from the user. The system operates entirely offline, ensuring complete privacy by keeping all data on the device. It captures screenshots periodically, segments them into meaningful parts, and stores them in a vector database for semantic retrieval. Lightweight algorithms and compressed models are used to guarantee smooth performance even on personal devices with limited hardware resources. Inferex offers a private, context-aware, and searchable memory of the user's digital interactions, making it a powerful tool for productivity, learning, and everyday information recall.

الملخص

إنفريكس (Inferex) هو تطبيق ذكي لسطح المكتب صُمم لمساعدة المستخدمين على تذكّر واسترجاع المعلومات التي شاهدوها سابقًا على أجهزتهم، سواء كانت من فيديو، مستند، أو موقع إلكتروني، دون الحاجة إلى حفظها أو تنظيمها يدويًا. في عالمنا الرقمي اليوم، يتفاعل المستخدمون يوميًا مع كميات ضخمة من المعلومات، ومع ذلك فإن تذكّر التفاصيل الدقيقة من الأنشطة السابقة لا يزال تحديًا مستمرًا. تعتمد معظم أدوات الذكاء الاصطناعي المتاحة حاليًا على أن يقوم المستخدم بتوفير سياق السؤال يدويًا للحصول على إجابة دقيقة وشخصية، مما يقلل من كفاءتها ويجعل استخدامها أقل سهولة. يحلّ إنفريكس هذه المشكلة من خلال إنشاء سياق محلي تلقائي وخاص بناءً على نشاط الشاشة لدى المستخدم، مما يمكّنه ليس فقط من استرجاع المحتوى الذي تم عرضه سابقًا، بل أيضًا من الإجابة على الأسئلة المتعلقة به بذكاء ودون الحاجة إلى إدخال أي معلومات إضافية. يعمل النظام بشكل كامل دون اتصال بالإنترنت، مما يضمن خصوصية المستخدم من خلال إبقاء جميع البيانات محفوظة على الجهاز فقط. يلتقط التطبيق لقطات شاشة بشكل دوري، ويقوم بتقسيمها إلى أجزاء ذات معنى، ثم يخزنها في قاعدة بيانات يمكن البحث فيها بطريقة دلالية. كما تم استخدام خوارزميات خفيفة ونماذج مضغوطة لضمان أداء سلس حتى على الأجهزة الشخصية ذات الموارد المحدودة. يقدم إنفريكس ذاكرة رقمية خاصة، مدركة للسياق، وقابلة للبحث، مما يجعله أداة قوية لزيادة الإنتاجية، وتسهيل التعلّم، واسترجاع المعلومات في الحياة اليومية.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my **parents**, whose unwavering love, support, and sacrifices have made everything I've achieved possible. The hardships they endured and the strength they showed through every challenge laid the foundation for the person I am today. Without their guidance, belief in me, and countless acts of selflessness, I would not be where I am now. This accomplishment is as much theirs as it is mine.

I would also like to extend my heartfelt gratitude to my teammates, **Mostafa Elsayed**, **Ali Abd-Elmotalb**, and **Mohamed Maher** for their dedication, hard work, and unwavering support throughout this endeavor. Their contributions have been invaluable, shaping our success with their expertise and commitment. Thank you for being an indispensable part of this journey.

I also want to thank **Dr. Sandra Wahid** for her clear guidance, helpful feedback on each design version, and timely assistance whenever challenges arose. Her steady support was vital to this project's success.

Table of Contents

Inferex **1**
ACKNOWLEDGMENTS **4**
Table of Contents **5**
List of Figures **6**
List of Tables **7**
List of Abbreviations **8**
Contacts **9**
Chapter 1: Introduction **11**
1.1 Input Processing Module: Handling Image Inputs 11
1.2. Extractive Question Answering Module: Improving System Accuracy 11
LLM Model Compression Module:
Compressing using Huffman encoding 12
Chapter 2: System Design and Architecture **13**
2.1 Input Processing Module 13
2.1.1 Image Embedding 14
2.1.2 Vector Database Management 14
2.1.3 Retrieval and Reconstruction 14
2.2 Extractive Question Answering 15
2.3 LLM Model Compression 17
Chapter 3: Conclusions and Future Work **23**
3.1. Faced Challenges 23
3.2. Gained Experience 23
3.3. Conclusions 23
References **24**

List of Figures

Chapter 2: System Design and Architecture..... 13
Figure 2.1: Input processing module..... 13
Figure 2.2: Huffman Tree of encoded exponent bit..... 18
Figure 2.3: Decoding process using LUTs..... 19
Figure 2.4: Hardware consumed while using the uncompressed model..... 20
Figure 2.5: Hardware consumed while using the compressed model..... 20

List of Tables

Chapter 2: System Design and Architecture..... 13
Table 2.1: Machine Learning Approaches (Accuracy)..... 16
Table 2.2: Deep Learning Approaches (Accuracy)..... 16

List of Abbreviations

AI: Artificial Intelligence
Bi-LSTM: Bidirectional LSTM
BoW: Bag of Words
LLM: Large Language Model
LSTM: Long Short-Term Memory
LUT: lookup table
QA: Question Answering
SVM: Support Vector Machine
CPU: Central processing unit
GPU: Graphics processing unit
OS: Operating System
RAM: Random Access Memory
NLP: Natural language processing

Contacts

Name	Email	Phone Number
Waleed Ousama Khamees	walid.osama.khamees@gmail.com	+2 01029356285

Supervisor

Name	Email	Number
Sandra Wahid	sandrawahid@hotmail.com	-

This page is left intentionally empty

Chapter 1: Introduction

In this project, I contributed to three core modules: the Input Processing Module, the Extractive Question Answering Module, and the LLM Model Compression Module. My work was focused on supporting the team's goals of enhancing the system's capabilities, from handling new data inputs to improving the performance and efficiency of the core language model.

1.1 Input Processing Module: Handling Image Inputs

Within the Input Processing Module, my main responsibility was to help develop a pipeline for handling image-based inputs. The objective was to allow visual data to be smoothly integrated into the system for effective indexing and retrieval. This involved contributing to a mechanism that could process raw images. I worked on the system for normalizing various image formats and sizes to ensure a standardized input for downstream processes. A significant part of my focus was on the indexing strategy, which involved converting processed image data into embeddings that could be queried more easily. This was aimed at reducing latency during retrieval and helping the system manage a large volume of visual information.

In addition to image processing and indexing, I was also responsible for managing how the database handled all data within the system. This included designing and implementing efficient strategies for inserting, updating, and organizing data in the database to support high-performance access and retrieval. To enhance the system’s scalability and responsiveness, I also worked on chunking strategies—breaking down large datasets or inputs into smaller, manageable units. These chunks facilitated more efficient processing, storage, and querying, and were particularly valuable in improving throughput and ensuring consistent performance under heavy data loads.

1.2. Extractive Question Answering Module: Improving System Accuracy

In the Extractive Question Answering Module, my efforts were aimed at helping to improve the system's accuracy and performance. I worked on implementing a question classification model to serve as a preliminary step in the pipeline. This classifier helps categorize incoming user queries, allowing the system to select a more suitable strategy for answer extraction. For example, questions identified as factual could be routed to a more precise extraction model.

Additionally, I explored and helped implement techniques for "answer injection." This approach involved providing the extractive model with contextual clues about the expected answer type based on the question's structure. By doing so, we could help constrain the search space for the answer within the text. These modifications contributed to an overall improvement in the precision of the answers provided by the system.

LLM Model Compression Module: Compressing using Huffman encoding

For the LLM Model Compression Module, my primary responsibility was to design and implement an algorithm to compress the model. I developed this compression algorithm specifically to run on a CPU, focusing on efficiently encoding the model's parameters into a more compact format.

In addition to creating the compression logic, I was also responsible for developing the corresponding decompression script. This script was designed to reconstruct the original high-precision weights from their compressed state.

A critical part of my work was to rigorously test and validate this entire pipeline. I executed the decompression algorithm on the CPU and then conducted a detailed, value-by-value comparison between the reconstructed weights and the original weights. The successful outcome of these tests confirmed that my compression and decompression algorithms were functioning correctly and were perfectly lossless, ensuring that the integrity of the model was maintained while achieving a significant reduction in its size. This validation was essential before the compression technique could be approved for use in our deployment workflow.

Chapter 2: System Design and Architecture

2.1 Input Processing Module

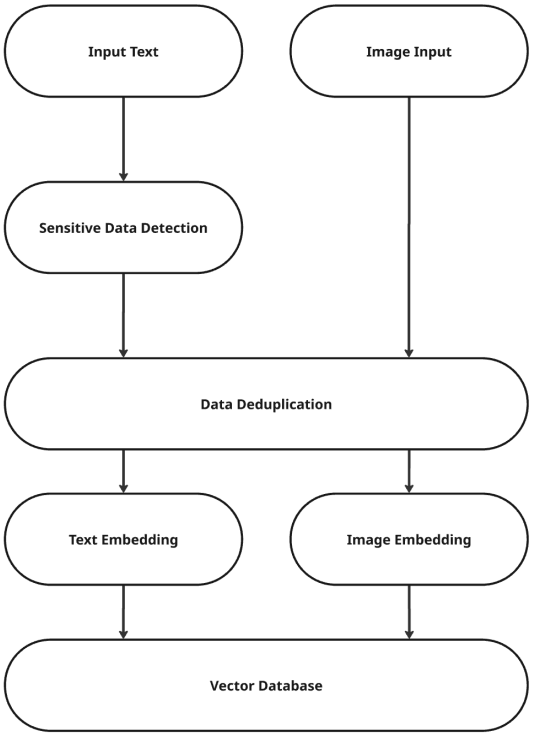

Figure 2.1: Input processing module

The Input Processing module is the main entry point for data in our system. It controls how we process and store both text and images. My role in this module was to manage how images are handled and inserted, and to design how all data is chunked and queried.

2.1.1 Image Embedding

The Image Embedding sub-module converts visual data into vector embeddings. It uses a pre-trained ResNet-50 model, which acts as a powerful feature extractor to understand the contents of an image. Here's the process:

- **Preprocessing:** The input image is resized to a standard 224x224 pixels, converted into a tensor, and then normalized. This prepares it for the model.
- **Feature Extraction:** The prepared tensor is passed through the convolutional layers of the modified ResNet-50.
- **Output:** The model outputs a high-dimensional vector that serves as the image embedding. This vector is a numerical representation that captures the visual features and meaning of the image.

2.1.2 Vector Database Management

Once an embedding is created, it is managed and stored in our Vector Database, which uses ChromaDB. For text, we use specific strategies to ensure high semantic precision:

- **Chunking Strategy:** We break long paragraphs into smaller, focused chunks. This step prevents the embeddings from becoming too general or "noisy."
- **Metadata Association:** Each chunk is stored with essential metadata: a **sequential index** to maintain the original order, and a unique **parent identifier** that links all chunks from the same paragraph together.

2.1.3 Retrieval and Reconstruction

This careful storage process makes our Retrieval and Reconstruction process possible. When a query is made:

- The system retrieves the single most relevant text chunk for the query.
- It then uses the parent identifier from the metadata to fetch all sibling chunks.
- Finally, it programmatically reconstructs the full, original paragraph.

This ensures the downstream question-answering module receives complete context, allowing it to provide more accurate responses.

2.2 Extractive Question Answering

Extractive Question Answering is a task in Natural Language Processing where a model identifies the answer to a question directly from a given text. Instead of generating a new response, the model extracts the precise span of text—a word or sequence of words—that contains the correct information.

For example, given the following context:

> **Context:** "The Amazon rainforest is the largest tropical rainforest in the world, covering an area of approximately 6.7 million square kilometers. It is home to an estimated 390 billion individual trees divided into 16,000 species."
>
> **Question:** "How many individual trees are in the Amazon rainforest?"

The model would extract the correct answer directly from the text:

> **Answer:** "390 billion"

My primary goal was to experiment with methods to improve the performance and accuracy of my extractive QA system. My central hypothesis was that I could enhance the model's ability to locate the correct answer by first classifying the *type* of question being asked.

---

2. Experiment: Enhancing the QA Model with a Question Classifier

I hypothesized that if the main QA model knew what type of entity the answer should be (e.g., a person, date, or location), it could more effectively locate the correct text span. This led to a multi-stage experiment.

Stage 1: Building a High-Accuracy Question Classifier

A prerequisite for this approach was a robust question classifier. I used the well-known **TREC dataset**, which categorizes questions into six types: Abbreviation, Location, Person, Numeric, Entity, and Description.

To find the best-performing model, I benchmarked several machine learning and deep learning approaches:

- **Machine Learning Models:** I tested Support Vector Machine (SVM) and Logistic Regression, using a variety of features: Bag of Words (BoW), Bigrams, TF-IDF, and rich embeddings from BERT.

- **Deep Learning Models:** I implemented both a standard Long Short-Term Memory (LSTM) network and a Bidirectional LSTM (Bi-LSTM).

Classifier Performance:

The results, shown below, indicate that the combination of **BERT embeddings with a Logistic Regression model** achieved the highest accuracy at **93.12%**. This made it the strongest candidate for integration into my QA pipeline.

Table 2.1: Machine Learning Approaches (Accuracy)

Feature/Model	SVM	Logistic Regression
Bag of Words	88.07%	86.05%
Bi-gram	88.68%	87.87%
TF-IDF	89.89%	86.05%
BERT Embeddings	92.92%	93.12%

Table 2.2: Deep Learning Approaches (Accuracy)

Model	Performance
LSTM	86.60%
Bidirectional LSTM	84.20%

Stage 2: Integrating the Classifier into the QA Model

With a highly accurate classifier developed, I explored two methods for integrating its output into the primary QA model:

1. **Direct Injection:** The predicted question type was inserted as a special token directly into the model's input sequence: `[CLS] Question_Type [SEP] Question [SEP] Context [SEP]`.
2. **Side-Task Learning:** The standard input format (`[CLS] Question [SEP] Context [SEP]`) was maintained. However, I added an auxiliary prediction layer where the model was concurrently trained to predict the question type from the `[CLS]` token's output. The loss from this side task was then combined with the main QA loss.

3. Experimental Outcome and Conclusion

Despite the high accuracy of the standalone classifier, the integration experiments revealed only a marginal impact on the QA model's performance.

Ultimately, while this was a valuable exploration, the question classification component was not included in the final module design. The minor F1 score improvements did not justify the significant increase in system complexity and computational overhead, which were key constraints of my project.

2.3 LLM Model Compression

The LLM Compression module I designed allows users with powerful commercial GPUs to take full advantage of them. It enables the use of original models with full weights, without quantization, preserving the highest possible quality. This approach also lowers the hardware specifications required to run these LLM models, as the memory needed for a full, uncompressed model is often much higher than that for a compressed one.

After we analyzed the model, we discovered that the exponent has a non-uniform distribution; the information included in the exponent bits can be represented with an average of 2.6 bits.

To compress the exponent, I applied Huffman encoding, a widely used lossless data compression algorithm based on the frequency of symbols. Huffman coding assigns shorter binary codes to more frequent values and longer codes to less frequent ones, minimizing the average number of bits required. Since the exponent values in the model weights follow a non-uniform distribution, encoding the most common values with fewer bits significantly reduced the overall storage cost of the exponent field.

This technique allowed us to reduce the average bit length of the exponent from 8 bits to approximately 3 bits, contributing substantially to the overall model compression.

Traditional Huffman decoding works by traversing the Huffman tree bit by bit, an inherently sequential process ill-suited for GPUs. To overcome this, I built a prefix-free lookup table (LUT) to replace the tree traversal. This table enables a direct mapping from bit patterns to decoded exponent values in a single lookup, facilitating highly parallel decoding across GPU threads and dramatically improving decoding speed.

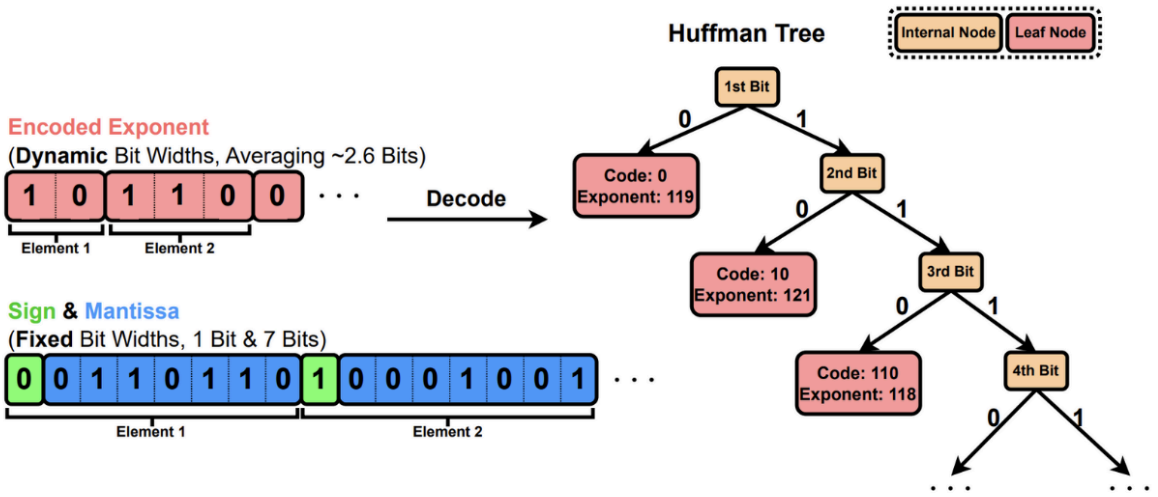

Figure 2.2: Huffman Tree of encoded exponent bit

The size of the LUT is 2^L, where L is the maximum bit length of any Huffman code. The decoding process is as follows:

1. Read the next L bits from the compressed bitstream.
2. Use the LUT to retrieve the corresponding decoded exponent.
3. Consult a second table that stores the actual bit length of each exponent's Huffman code.
4. Advance the bitstream pointer by that length.
5. Repeat the process for the next symbol.

This method allows for fast, parallel decoding without branching. However, the LUT can become excessively large. For example, a maximum Huffman code length (L) of 32 would require a LUT with 232 entries, which is over 4.29 billion entries and extremely memory-intensive.

To address this, I divided the LUT into four disjoint lookup tables, each of size 28. With each entry being one byte, the total space for these tables is 4×28=1024 bytes. Additionally, I maintain a code length LUT of size 28=256 bytes. The total memory usage for these tables is 1024+256=1280 bytes, which is small enough to fit into GPU shared memory for fast access.

The decoding process using this multi-level LUT structure proceeds as follows:

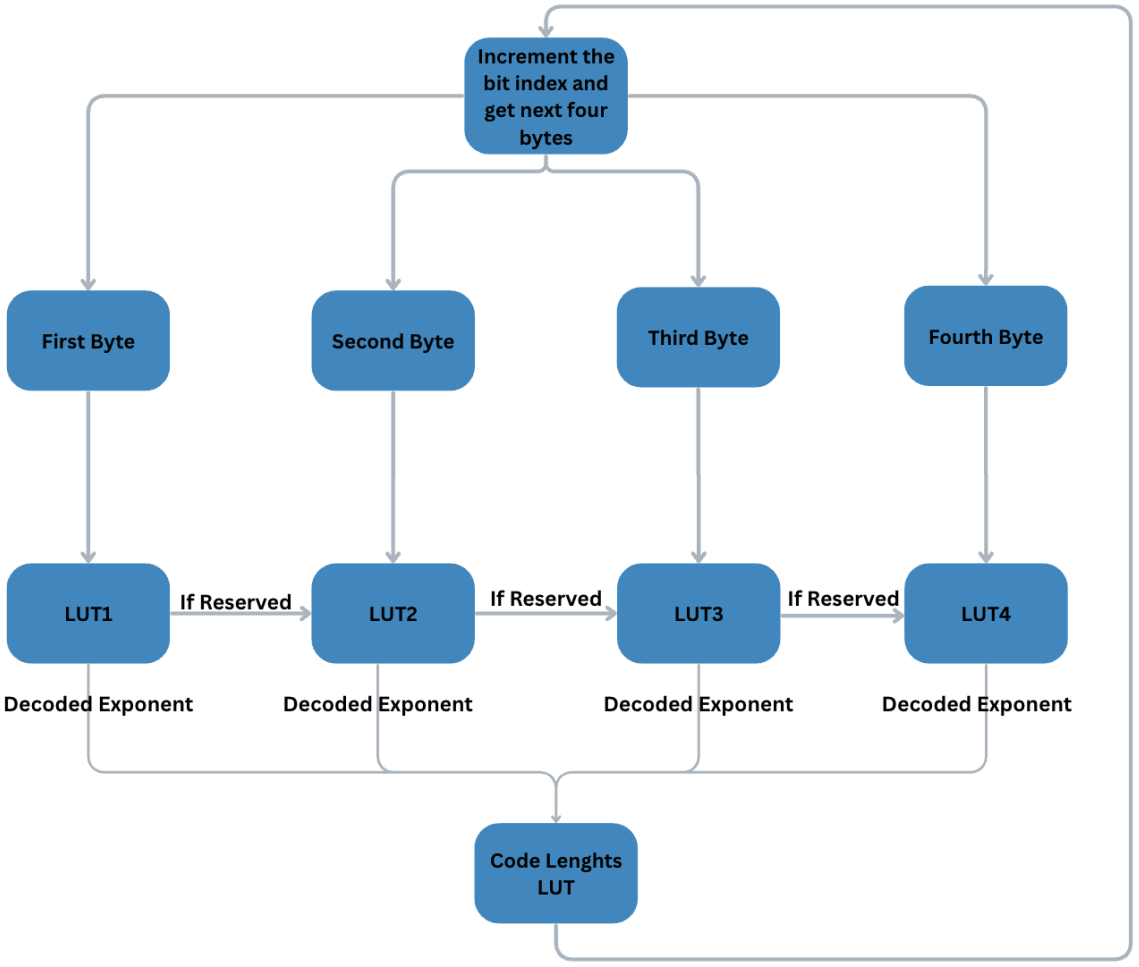

Figure 2.3: Decoding process using LUTs

1. Read the first byte from the bitstream and use it to index into LUT1.
2. If the result is a reserved marker (indicating a multi-byte code), read the second byte and index into LUT2.
3. Continue this process through LUT3 and LUT4 if necessary, reading one additional byte at each step.
4. Once the exponent is decoded, access the code lengths LUT to find the actual length of the Huffman code.
5. Advance the bitstream pointer by that length and proceed to the next symbol.

For parallel decoding, I assign 8 bytes of the compressed bitstream to each thread. A challenge here is that Huffman codes have variable lengths, so a thread's starting position may not align with the beginning of a valid code. To solve this, I introduced

an auxiliary array called `gaps`, which provides each thread with the correct bit offset to start decoding. Each entry in the `gaps` array only requires 5 bits, which is sufficient to cover the maximum possible offset of 31 bits.

Another challenge is determining the correct output position for each decoded exponent. Storing a 32-bit offset for each thread would create significant memory overhead. To avoid this, I store only a single offset per thread block, which marks the starting output position for the first exponent decoded by that block. This requires additional logic within the kernel for each thread to calculate its specific output position, but greatly reduces memory usage.

Results on Qwen3-4B:

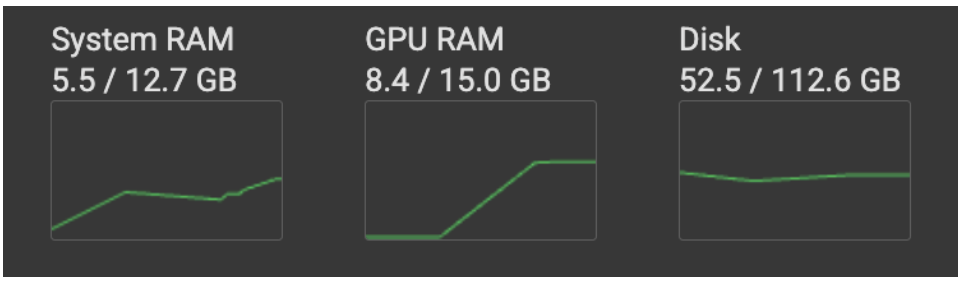

Figure 2.4: Hardware consumed while using the uncompressed model

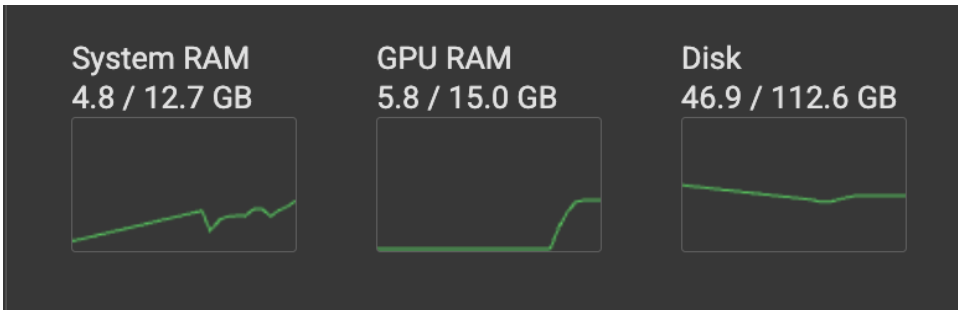

Figure 2.5: Hardware consumed while using the compressed model

The GPU memory usage for the uncompressed model is 8.4 GB, while the compressed version uses only 5.8 GB, resulting in a 30% reduction in memory consumption. This significant decrease demonstrates the effectiveness of our compression approach in reducing resource requirements without compromising model functionality.

Chapter 3: System Testing and Verification

In this chapter, I will demonstrate how I tested and verified the correctness and effectiveness of my contribution

3.1. Testing Setup

The testing was conducted on a laptop with the following specifications:

- CPU: AMD Ryzen 7 5800H
- RAM: 16 GB
- GPU: NVIDIA RTX 3060
- OS: Windows 11
- Python version: 3.10

3.2. Testing Plan and Strategy

For the AI-based modules, the strategy involves splitting the data into training and validation sets, monitoring the loss on both, and selecting the model version that achieves low validation loss without signs of overfitting.

For the Compression module, the approach is to compare the original weights with the decompressed weights to ensure the compression and decompression processes preserve accuracy.

3.2.1. Module Testing

Input Processing Module: the image processing module was tested manually to ensure that similar images were grouped together using cosine similarity as a metric for measuring how close the images are to each other.

Extractive Question Answering: The models were trained on the same data and then validated on a never-before-seen dataset to measure their accuracy.

In The model Compression module: we implement a encode and decode function that run on the cpu since the data compression was done on the cpu we wanted to ensure that the decompressed exponent were the same as the original exponent so each time we encoded a layer and we compared against the original weights to ensure that they’re the same and that the compression is lossless.

3.3. Testing Schedule

Testing was conducted immediately after the completion of each component to ensure its functionality and correctness before proceeding to the next stage

Chapter 3: Conclusions and Future Work

3.1. Faced Challenges

A significant challenge arose during the implementation of the model compression module. When constructing the Lookup Tables (LUTs) for Huffman decoding, we encountered mapping conflicts where multiple elements were assigned to the same position. This rendered the LUTs non-functional. The issue was resolved through an iterative process of manually adjusting the frequencies of conflicting elements until a stable, conflict-free table was achieved.

In addition to implementation, validating the compression process was also demanding. We needed to ensure that the encoding and decoding processes were fully consistent and produced identical outputs. This verification process was time-consuming, even for relatively small LLMs, due to hardware limitations that constrained testing speed and scale. Nonetheless, achieving reliable and lossless compression was critical to the overall success of the system.

3.2. Gained Experience

This project provided deep practical experience in several key domains. I developed a strong understanding of Natural Language Processing (NLP) tasks, particularly the techniques for lossless compression of Large Language Models (LLMs). Furthermore, I gained hands-on expertise in applying various machine and deep learning models to solve complex problems in natural language processing.

3.3. Conclusions

In conclusion, this project successfully demonstrated two key achievements. First, we implemented a method to losslessly compress a Large Language Model, preserving its full accuracy while reducing its size. Second, we achieved a marginal enhancement in the performance of the Question Answering module by implementing a preceding machine learning model to classify and constrain the expected answer type.

References

[1] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of the I.R.E.*, vol. 40, no. 9, pp. 1098–1101, 1952.

[2] T. Zhang, Y. Sui, S. Zhong, V. Chaudhary, X. Hu, and A. Shrivastava, “70% Size, 100% Accuracy: Lossless LLM Compression for Efficient GPU Inference via Dynamic-Length Float,” arXiv preprint arXiv:2504.11651, 2025.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” arXiv preprint arXiv:1810.04805, 2019.